# OS-3: The Oregon State open shop operating system

*by* JAMES W. MEEKER, N. RONALD CRANDALL, FRED A. DAYTON, and G. ROSE

*Oregon State University Computer Center*
Corvallis, Oregon

## INTRODUCTION

This paper is a discussion of the OS-3 operating system developed at Oregon State University. Before proceeding to a discussion of that system, it is appropriate to say a few words in order to view this work within a more global context.

It is little more than a truism to say that computers are difficult and expensive to use. That is to say, computers are difficult and expensive with respect to the problems men wish to solve. One primary reason for this state of affairs is embedded in many years of cultural history. In the absence of computing machinery we have developed methodologies that ingeniously avoid the necessity for computational solutions.

For example, if it is necessary to perform several million multiplications in order to test a hypothesis, then until recently it was quite likely that such a hypothesis would remain unexplored. Now, of course, this situation is radically changed. Nevertheless, the thinking that will take proper advantage of current computer power is still in its infancy. Thus, computers are difficult and expensive to use because we haven't yet learned how to use them.

If we could see clearly enough into the future to determine those approaches to problem solving that will be most successful during the coming decades, then we would not hesitate to develop software tailored to best underwrite these approaches. To the degree that we lack the foresight to proceed in this fashion, there remains an acceptable alternative: the general purpose computer utility. If we can provide a utility that is inexpensive, reliable, and convenient to use, then we can deliver a powerful tool directly into the hands of the problem solver.

One overall requirement implicit in this idea is that such a utility must be comprehensive enough to free the problem solver from the burden of becoming a system's programmer. This burden has been responsible for the migration of many scientists from other disciplines into systems software development after which they proceed to neglect half a lifetime of training in their own field: an expensive proposition.

In addition, our objectives in developing a computer utility include the following: the utility should be

1. inexpensive—that is, system overhead should be small.
2. convenient—programming conventions should be easy to learn and use as well as generally accessible.
3. transparent—the user should have ready access to information about the state of the system, his account with the system, the status of a running program, and the contents of his saved storage.
4. information oriented—facilities must be available for creating, manipulating, and maintaining files of arbitrary structural complexity.
5. self extending—facilities should be available for building upon past experience in a facile way.

This paper is specifically a discussion of a time-sharing operating system that is intended to satisfy the first three of these objectives. The remaining two are also under development at O.S.U., but do not exist at the level of the resident operating system.

The following sections include a description of the system as seen by a user, a brief discussion of salient software characteristics, and a summary of system performance.

### *User features*

OS-3 is a time-sharing operating system for the Control Data 3300 computer. It was developed at O.S.U.

and is presently our principal operating system. Currently the system can service up to 32 on-line users together with a batch user, and during early 1969, it is anticipated that this number will exceed fifty.

The system is used by various departments on campus, local industry, and other colleges throughout the State of Oregon. At the present moment the system logs approximately 4,000 console hours per month.

The operating system multiplexes available hardware resources among concurrent users (CRT, Teletype, and batch) in a time-slicing fashion. Processor time and core memory are allocated to running programs based upon considerations of program demand and page traffic flow.

### Hardware environment

In order to orient those not immediately familiar with the 3300, it should be sufficient to say that the machine includes the following features:

- 24-bit word
- Paged memory and page file
- Executive mode operation
- Usual interrupt system
- Usual register configuration
- Real time clock
- 64-word fast core

Memory is expandable in units of $2^{14}$ words; we possess four such units. The 3300 addressing scheme will permit a user's program to address at most $2^{16}$ words. Our hardware configuration is depicted in Figure 1.

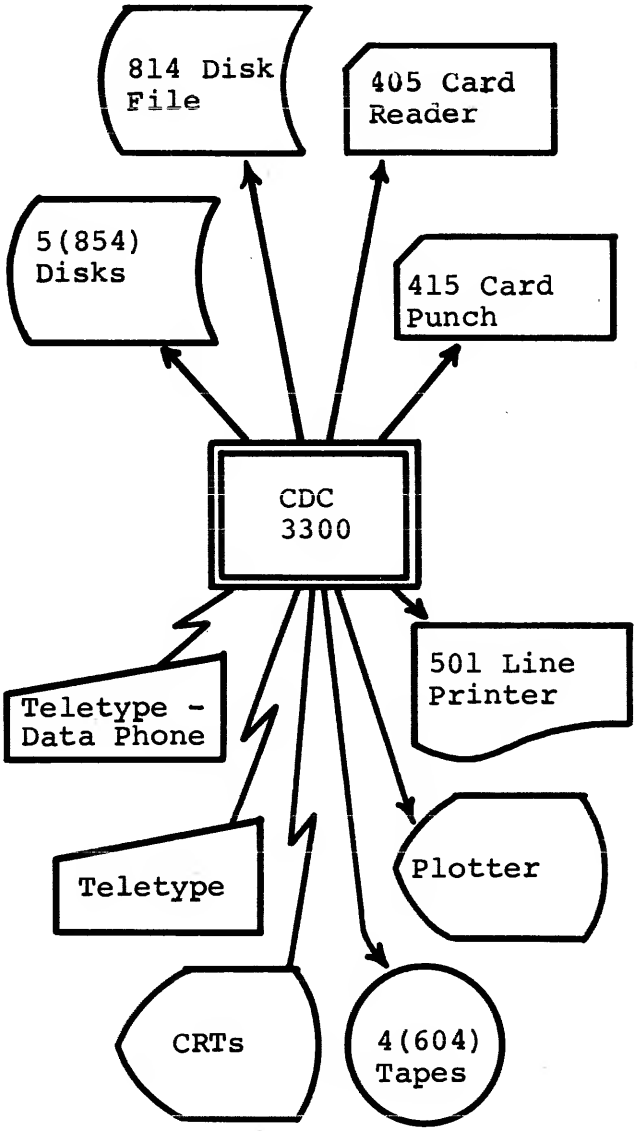



Figure 1—Hardware configuration

### System library

The system library is composed of an absolute library and a user generated library. Most of the programs in the absolute library are written in reentrant code and treated as such by the operating system. The library includes:

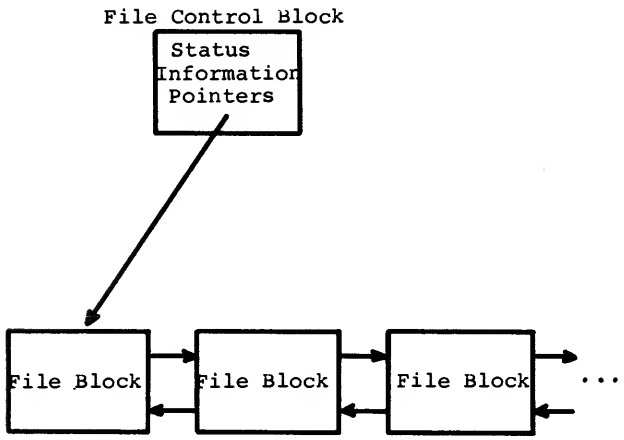
1. Fortran compiler—a modified CDC Fortran with several input/output options including short form diagnostics suitable for listing at a console.
2. Algol compiler—modified CDC Algol.
3. OSCAR—an O.S.U.-developed conversational arithmetic interpreter with stored program capabilities.[1] OSCAR recognizes scalars, vectors, matrices; it is fully recursive and allows definition of functions and abbreviations. Formatting is allowed but due to the use of default format options it is not required. OSCAR can communicate with other available languages.
4. Compass—an extended version of the 3300 assembly language.
5. RADAR[2]—an on-line debugging language. RADAR includes an assembler/disassembler and permits single stepping through a program. (A CRT oriented version of the language is also available.)
6. Edit[3]—an on-line editing language with context searching capabilities.
7. Sort/Merge
8. Utilities—including Autoloading and file manipulation.

The user-generated library contains programs local to each user. If a user declares a program to be public,

then anyone can access it in a fashion analogous to that used for the absolute library. Of course, only the creator can modify the program, and that only when it is not currently in use.

### Input/output

At the level of the operating system, files have little structure. Two types of files exist: linear and random access. A *linear file* is physically a series of fixed length blocks. These are dynamically allocated up to some storage limit that is preset for each user. A user reads and writes a linear file in variable length records; writing a record in the middle of a linear file causes the remainder of that file to be released (see Figure 2).




File Control Block

| | |
|---|---|
| Load Point | |
| Core Pointer | |
| Block number of current block | |
| Current Position Pointer | |
| Status | Number of blocks beyond current block |
| Hardware Type | End Position Pointer |
| Total Length | |
| Accounting Word | |

File Block

| |
|---|
| Forward Pointer |
| Backward Pointer |
| Wordcount |
| Record |
| Wordcount |
| Record |
| . . . |

Figure 2—Linear file

*Random access files* can be viewed as terminally open files together with a file pointer. This type of file is essentially a large block of apparently contiguous storage (maximum size = $(2^9 - 2)2^{18}$ words). Physically the random access file is a series of fixed length blocks with a two level directory. The topmost directory holds pointers to a set of directory blocks, each of which contains pointers to a set of linked storage blocks. A serial pass through the file can use the links for traversing blocks, while a random search utilizes the directories (Figure 3). A user reads or writes a random access file n words at a time beginning at the current pointer location. At the end of the operation, the pointer is advanced by n words.

Files may be named and saved in semi-permanent storage. It is also possible to create temporary files that can be assigned to a logical unit number (in the range 0-99). Any saved file can be equated to a logical unit and conversely, a logical unit may be subsequently saved under a file name. While running programs manipulate logical units, not named files, the major programs in the library allow the user to supply only a file name that is then assigned to a logical unit by the program. Any file may be file protected. Any file may be placed in the public domain by preceding its name by an asterisk (*).

In addition to being equated to files, logical units may assume the following hardware types:

- line printer
- card reader
- card punch
- console input
- console output
- plotter
- magnetic tape
- null

### Request processor

A console user is placed in control mode at the time he logs in, and he can revert to this mode at any time. Communication with the system while in control mode takes place via the *request processor*. In this language the user can call for any of the supported systems, execute jobs, manipulate files, examine the state of the system, the status of his running program, inspect information about his account, etc.

Of special note is that a user with a program in execution can cause that program to be suspended for an indefinite time by reverting to control mode. From control mode it is then possible to execute a sequence of commands and subsequently resume running by typing the command 'GO'. This capability may be used

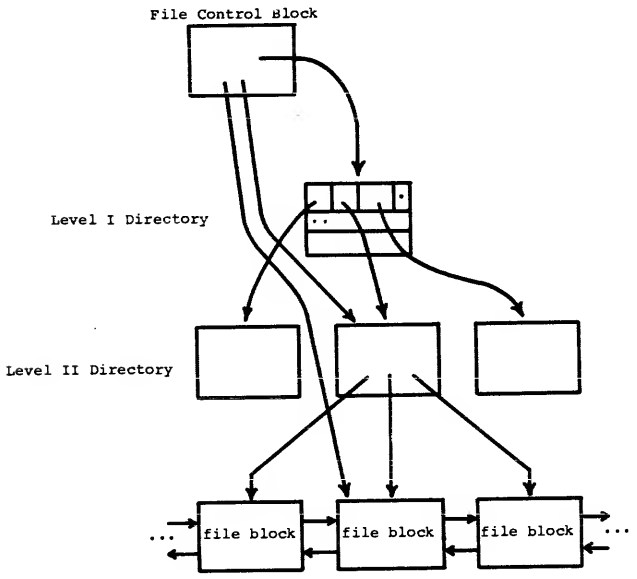



Figure 3—Random access file

to allow the user to solicit information about the running program or accomplish on-line error recovery. In a similar way, the command 'MI' simulates a manual interrupt that may be used for communication between the request processor and other supported systems.

## Accounting

Associated with every user is a unique job/user number pair. The first number is used for billing purposes; the second is a code used for identification. Associated with each job/user number are three limits:

1. Maximum time—a time limit equal to the total processor time that may be used. This number is appropriately decremented at the conclusion of a session at the console.
2. Saved file space—a maximum storage limit for saved files.
3. Scratch file space—a maximum storage limit for temporary files.

When a user logs into the system, his time is automatically set to one minute and his scratch file limit to 100 storage blocks. He may change these limits up to the maximum limits associated with his job/user number.

When a console user logs off or when a batch job is completed, the charges for that job are converted to a CPU time equivalent and deducted from the user's remaining total time.

Charges are made for CPU time, some I/O, elapsed time at the console, and saved storage space. Since system overhead increases with increasing demand on the system, the apparent CPU time required for a job will be higher during peak hours. A user can get an indication of the current system loading by typing the command 'TRAFFIC'. If he decides against running and logs off at this point, no charge is incurred.

### *System characteristics*

Several guidelines were adopted in writing the system. Development has been modular with all modules written strictly in tightly coded assembly language. Since core is at a premium in our configuration, it was imperative to keep the size of the resident monitor at a minimum. To this end, even the request processor is subject to memory swapping. In addition, the system is hardware sensitive, i.e., particular I/O instructions in the order code have been avoided because their failure rate, over the years, has been disproportionately high. Further, the system is highly parameterized. For example, if a memory module fails, a parameter can be changed and the system run in reduced memory.

With these guidelines in mind, the system can be described roughly as follows. Associated with every active user is a fixed length *program status area* (PSA) together with a set of attached lists. PSA's are linked together into a *user queue* (Figure 4). In general, OS-3 runs under the influence of this user queue, with resource allocation and I/O dependent upon the list structures connected to a particular PSA. At any given

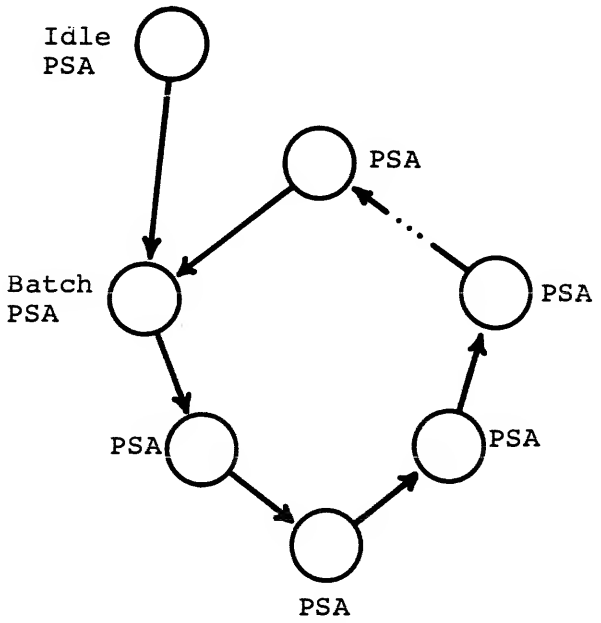



Figure 4—User queue

moment, the system is running a single user whose PSA is indicated by a pointer. Alteration of the contents of this pointer occurs at the end of discrete time intervals called quanta (see Figure 5). This general picture is qualified by other considerations as will be seen.

Physical memory is partitioned into four distinct areas:

1. The resident monitor
2. Free storage
3. File core blocks
4. Available memory

The resident monitor occupies approximately 10K words of core. PSA's with associated lists and console I/O occupy space in free storage that can be dynamically expanded. Two pages ($2^{11}$ words per page) of file core blocks are reserved for disk transfers. The remaining core is assigned to running users and swapped. Any user can work in an address space of up to $2^{16}$ words of virtual memory.

The system includes the following major modules:

- Scheduler —allocates processor time and memory
- User I/O —interprets and handles user generated requests for I/O
- Intsort —a recursive interrupt processor
- I/O Drivers —a set of integrated device drivers
- Request Processor —the command language interpreter
- Accounting —user accounting routines

This paper will consider only the scheduler and User I/O in greater detail.

## User I/O

User I/O is a collection of routines that supervise the peripheral device drivers and user requests for input or output operations. This module is divided into two principal sections: the mass storage device scheduler and the executive request interpreter.

The mass storage (M.S.) device scheduler governs a multi-priority transfer queue containing mass storage I/O requests. This queue is created by the M.S. device scheduler in response to requests received from peripheral devices in operation as well as user programs. For example, when the line printer driver exhausts its current block of output, and the next block is requested, the M.S. device scheduler interprets this request, assigns it a priority, and extends the transfer queue accordingly.

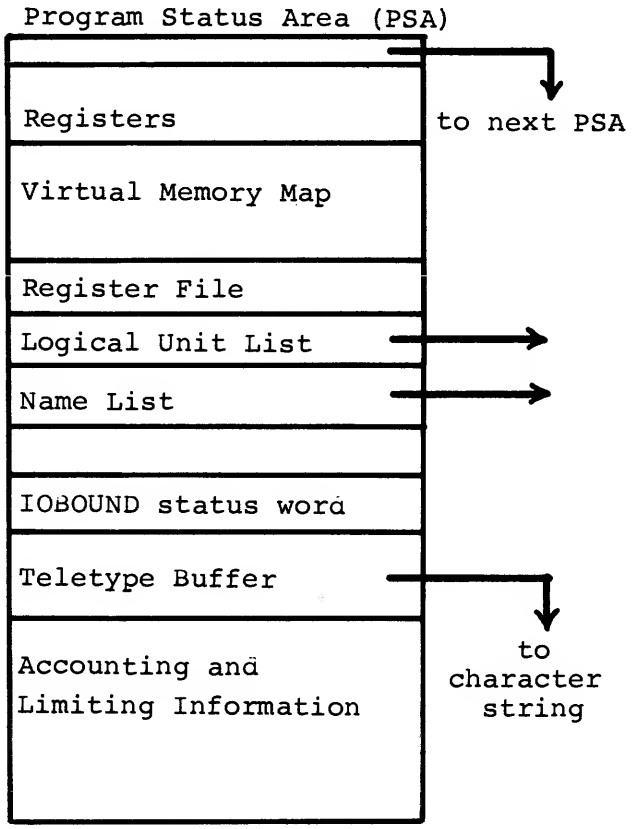



Figure 5—Program status area

The executive request processor interprets all executive requests (trapped instructions) generated by a user. After decoding a request, the processor delivers control to the appropriate routine. All I/O called for by a user is included in the category of executive requests.

## Time and memory allocation

Allocation of the time and physical memory occurs primarily within the *Scheduler* and is controlled by the tables PAGETABLE and PAGETIME. Memory is divided into pages by the hardware. Each page has a page number associated with it which is used for relocation and reference to the software tables. Each page of physical memory has one word in PAGETABLE and a corresponding word in PAGETIME. Part of each PAGETABLE word serves as an indicator of the status of the associated page; the remainder is an address of a *page access word* which is used to map one page of virtual memory into the associated physical memory page. PAGETIME has one word per page that indicates the time at which a user last referenced the page in question. (See Figure 6.)

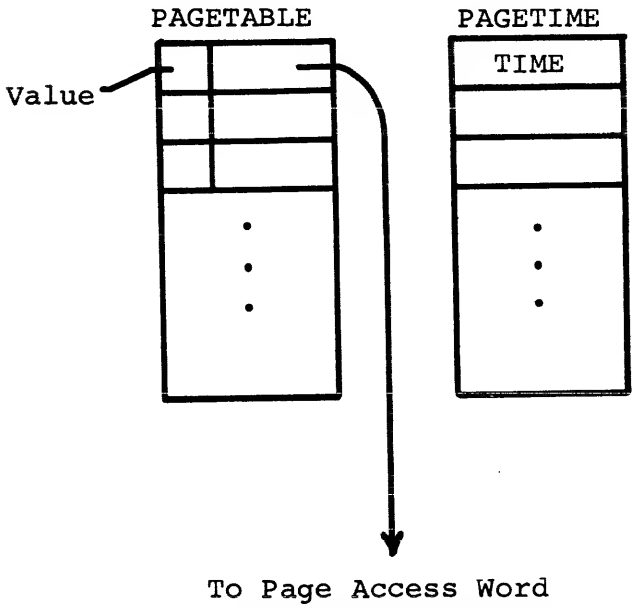



Figure 6—PAGETABLE and PAGETIME

When a page of memory is needed for swapping purposes, a search is performed on PAGETABLE to find the page with the smallest value. Value is infinite if the system bit is set or zero if the occupied bit is not set. In all other cases, a function is evaluated which is the sum of

$$\text{PAGETIME} - \text{clock} + \text{Value (I)} \qquad (1)$$

where

I is a set of status bits from PAGETABLE

clock is the current reading of the real time clock

and

Value is given by a function that maps bit configurations into time values.

PAGETIME is set to clock + 1 hour whenever a page is referenced and is shifted right one position every hour to prevent overflow. The values in PAGETIME do not age linearly because of this shift; however, the function is continuous and is not inaccurate in the region where the clock is set back one hour and PAGETIME entries shifted.

A user's reference to his virtual memory occurs upon detection of an illegal write interrupt, and the system is then table driven based upon the above tables and the virtual memory map (VMM) in the user's PSA. Each word in the VMM is either a page access word or a pointer to a page access word, depending upon whether the page in question is written in reentrant code (Figure 7).

Processor time is also allocated to each user by the Scheduler. A running program pointer (RPSAPTR) advances in a circle around the user queue. If a user's program is to be activated when RPSAPTR advances to his PSA, then the following condition must be satisfied:

$$\text{TIMELEFT} > 0 \,\&\, \sim \text{IOBOUND} \qquad (2)$$

where

TIMELEFT—is the time left in the current quantum

and

IOBOUND —is a status word in the PSA that indicates whether the user is awaiting the completion of an I/O operation.

If no PSA satisfies condition 2, the RPSAPTR advances until a PSA that is not IOBOUND is found and sets that program to run another quantum.

It is clear that a user who requires extensive swapping will contribute significantly to the flow of page traffic

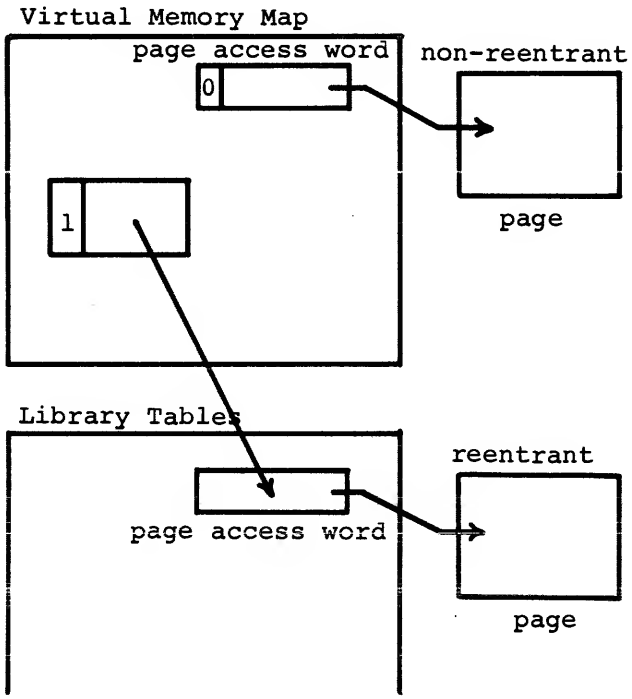



Figure 7—Page access word

to and from the disk. Moreover, such a user will receive poor service if the page traffic flow is heavy.

In order to cope with this situation, a more sophisticated scheduling arrangement is required. This kind of extension to the scheduling algorithm should be adaptive in nature, that is the system should recognize the existence of a problem situation and proceed to 'tune itself up.' Further, detection of the problem condition as well as modification of the scheduling tactics must be easily computable if an undesirable increase in system overhead is to be avoided.

At present, OS-3 includes an initial version of a demand scheduling strategy called the debogging algorithm. This algorithm governs the allocation of processor memory so as to minimize page traffic flow in the presence of varying user requests.

Conceptually, the algorithm can be viewed as a high priority pointer that is cycled around the user queue independent of the RPSAPTR. If a user is designated the high priority user, then his in-core pages increase in value, and he is automatically placed second in line in the swapping queue for requested pages.

In effect, the debogging strategy tends to delay users whose page requests are heavy with respect to current page traffic flow, and then run such users with greater priority for a period of time during which they can occupy a substantial amount of core.

In particular, the algorithm governs the behavior of the following independent categories of events:

1. Advancing the high priority pointer
2. Delaying troublemakers, and
3. Rehabilitating former troublemakers.

These categories are now described in greater detail.

The high priority pointer, HPP, is advanced to the next user whenever one of the following conditions is satisfied:

The user logs off (3)

The user becomes I/O bound (4)

$$WCT + (PR_i * 2^{K1}) \geq K2 \quad (5)$$

where

WCT —is the amount of wall clock time that has elapsed since the HPP was advanced

$PR_i$ —is the page request word in the $i^{th}$ user's PSA. $PR_i$ is incremented by one each time the $i^{th}$ user requests a page

and

K1, K2—are constants.

K1 influences the rate at which the HPP will shift to the next user if the current user is busy swapping. K2 is simply a constant that governs the cycle rate of the HPP.

If any of the conditions 3, 4, or 5 is satisfied, then

$$WCT: = \overline{SQ}: = QCT: = 0 \quad (6)$$

$$PR_i: = 0 \quad (i = 1, \ldots, n) \quad (7)$$

where

$\overline{SQ}$ —is the average length of the swapping queue

and

QCT —is the number of useful quantums of computing (i.e., time not spent in the idle loop).

The second category concerns the troublemaker. A user is a *troublemaker* if only the mass storage wait bit of his IOBOUND word is set, and if

$$PR_i \geq K3 \quad (8)$$

where

K3 —is a constant that determines the number of swap requests a user must generate in order to qualify as a troublemaker.

The *last troublemaker* is defined as the troublemaker that is located the greatest distance from the HPP in the direction of pointer rotation. A troublemaker is delayed by setting the delay bit in the IOBOUND word of his PSA. The last troublemaker will, in fact, be delayed if:

$$(SQ \geq K4) \ \& \ (\overline{SQ} \geq K5) \quad (9)$$

where

SQ —is a counter that contains the current length of the swapping queue

$\overline{SQ}$ —the average length of the swapping queue

K4 —is a stabilizing factor

and

K5 —determines heavy page traffic flow.

If condition (9) is satisfied then

$$SQ: = \overline{SQ}: = QCT: = 0 \quad (10)$$

The final category provides for the rehabilitation of former troublemakers. A *former troublemaker* is a user whose delay bit is set. The *closest former troublemaker* is defined to be the former troublemaker located the least distance from the HPP in the direction of pointer rotation. The closest former troublemaker may be rehabilitated by clearing his delay bit. This will occur when

$$(QCT \geq K6) \;\&\; (\overline{SQ} < K7) \qquad (11)$$

where

K6 —is a stabilizing factor

and

K7 —determines relatively light page traffic flow.

The effect of the preceding debogging strategy is to match available processor memory to user demands. If this cannot be done, then an obvious troublemaker is delayed, and, after a period of stabilization, the situation is sampled again to determine whether an acceptable match has occurred. If not, then another troublemaker is delayed, and so forth, until a match is achieved. Conversely, if user demands are not overloading the swapping queue, then former troublemakers are rehabilitated, one at a time. Of course, if several users require large quantities of physical memory, the recidivism rate will be high.

*System performance*

System performance measured in terms of system overhead tends to be quite good. If the total number of user hours for a month is compared to the total amount of billable CPU time for that period, it turns out that the system spends slightly more than 65 percent time in an idle loop. Of course, this might indicate that the system is heavily I/O bound; however, test measurements indicate that this is not the case.

In another test, switching time was measured by loading the system with a sample job mix. Jobs were chosen from three categories:

1. Compute bound
2. 65K swap bound
3. I/O bound

The 100 millisecond quantum was then reduced until no useful computing took place. This break-even point occurred at four milliseconds.

## ACKNOWLEDGMENTS

The authors are indebted to Steven K. Sullivan for his incisive comments, useful criticisms, and system programming support.

## REFERENCES

1 J DAVIS
*A brief description of OSCAR* (Second Revision)
OSU Computer Center cc–68–45

2 J MEEKER
*RADAR*
OSU Computer Center cc–68–30

3 F DAYTON W MASSIE
*OS-3 teletypewriter editor manual* (Revised)
OSU Computer Center cc–68–17